KNOWNSEC

Smart Contract Security Audit Report

Audit Results

PASS

★ ★ ★ ★ ★

Version description

Revised man	Revised content	Revised time	version	Reviewer
Yifeng Luo	Document creation and editing	2020/10/22	V1.0	Haojie Xu

Document information

Document Name	Audit Date	Audit results	Privacy level	Audit enquiry telephone
marlinprotocol Smart Contract Security Audit Report	2020/10/22	PASS	Open project team	+86 400-060-9587

Copyright statement

Company statement

Beijing Knownsec Information Technology Co., Ltd. only conducts the agreed safety audit and issued the report based on the documents and materials provided to us by the project party as of the time of this report. We cannot judge the background, the practicality of the project, the compliance of business model and the legality of the project , and will not be responsible for this. The report is for reference only for internal decision-making of the project party. Without our written consent, the report shall not be disclosed or provided to other people or used for other purposes without permission. The report issued by us shall not be used as the basis for any behavior and decision of the third party. We shall not bear any responsibility for the consequences of the relevant decisions adopted by the user and the third party. We assume that as of the time of this report, the project has provided us with no information missing, tampered with, deleted or concealed. If the information provided is missing, tampered, deleted, concealed or reflected in a situation inconsistent with the actual situation, we will not be liable for the loss and adverse impact caused thereby.

Catalog

1. Review

The effective testing time of this report is from October 20, 2020 to October 21, 2020. During this period, the Knownsec engineers audited the safety and regulatory aspects of marlinprotocol smart contract code.

In this test, engineers comprehensively analyzed common vulnerabilities of smart contracts (Chapter 3) and It was not discovered medium-risk or high-risk vulnerability,so it's evaluated as **pass**.

The result of the safety auditing: Pass

Since the test process is carried out in a non-production environment, all the codes are the latest backups. We communicates with the relevant interface personnel, and the relevant test operations are performed under the controllable operation risk to avoid the risks during the test..

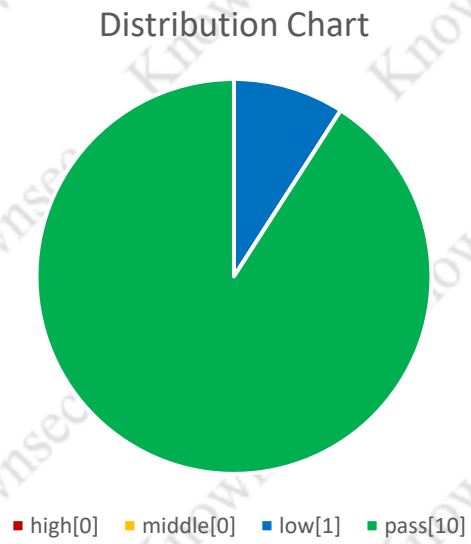
Target information for this test:

Project name	Project content
Token name	marlinprotocol
Code type	Token code
Code language	solidity
Code address	https://github.com/marlinprotocol/Contracts/tree/multi-delegate

2. Analysis of code vulnerability

2.1. Distribution of vulnerability Levels

Vulnerability statistics			
high	Middle	low	pass
0	0	1	10



2.2. Audit result summary

Other unknown security vulnerabilities are not included in the scope of this audit.

Result			
Test project	Test content	status	description
Smart Contract Security Audit	Reentrancy	Pass	Check the call.value() function for security
	Arithmetic Issues	Pass	Check add and sub functions
	Access Control	Pass	Check the operation access control
	Unchecked Return Values For Low Level Calls	Pass	Check the currency conversion method.
	Bad Randomness	Pass	Check the unified content filter
	Transaction ordering dependence	Low risk	Check the transaction ordering dependence
	Denial of service attack detection	Pass	Check whether the code has a resource abuse problem when using a resource
	Logic design Flaw	Pass	Examine the security issues associated with business design in intelligent contract codes.
	USDT Fake Deposit Issue	Pass	Check for the existence of USDT Fake Deposit Issue
	Adding tokens	Pass	It is detected whether there is a function in the token contract that may increase the total amounts of tokens
	Freezing accounts bypassed	Pass	It is detected whether there is an unverified token source account, an originating account, and whether the target account is frozen.

3. Result analysis

3.1. Reentrancy【Pass】

The Reentrancy attack, probably the most famous Blockchain vulnerability，led to a hard fork of Ethereum.

When the low level call() function sends tokens to the msg.sender address, it becomes vulnerable; if the address is a smart token, the payment will trigger its fallback function with what's left of the transaction gas.

Test results: No related vulnerabilities in smart contract code.

Safety advice：None.

3.2. Arithmetic Issues【Pass】

Also known as integer overflow and integer underflow. Solidity can handle up to 256 digits (2^256-1), The largest number increases by 1 will overflow to 0. Similarly, when the number is an unsigned type, 0 minus 1 will underflow to get the maximum numeric value.

Integer overflows and underflows are not a new class of vulnerability, but they are especially dangerous in smart contracts. Overflow can lead to incorrect results, especially if the probability is not expected, which may affect the reliability and security of the program.

Test results: No related vulnerabilities in smart contract code.

Safety advice：None.

3.3. Access Control【Pass】

Access Control issues are common in all programs,Also smart contracts. The famous Parity Wallet smart contract has been affected by this issue.

Test results: No related vulnerabilities in smart contract code.

Safety advice：None.

3.4. Unchecked Return Values For Low Level Calls【Pass】

Also known as or related to silent failing sends, unchecked-send. There are transfer methods such as transfer(), send(), and call.value() in Solidity and can be used to send tokens s to an address. The difference is: transfer will be thrown when failed to send, and rollback; only 2300gas will be passed for call to prevent reentry attacks; send will return false if send fails; only 2300gas will be passed for call to prevent reentry attacks; If .value fails to send, it will return false; passing all available gas calls (which can be restricted by passing in the gas_value parameter) cannot effectively prevent reentry attacks.

If the return value of the send and call.value switch functions is not been checked in the code, the contract will continue to execute the following code,and it may have caused unexpected results due to tokens sending failure.

Test results: No related vulnerabilities in smart contract code.

Safety advice：None.

3.5. Bad Randomness【Pass】

Smart Contract May Need to Use Random Numbers. While Solidity offers functions and variables that can access apparently hard-to-predict values just as block.number and block.timestamp. they are generally either more public than they seem or subject to miners' influence. Because these sources of randomness are to an extent predictable, malicious users can generally replicate it and attack the function relying on its unpredictablility.

Test results: No related vulnerabilities in smart contract code.

Safety advice：None.

3.6. Transaction ordering dependence【Low risk】

Since miners always get rewarded via gas fees for running code on behalf of externally owned addresses (EOA), users can specify higher fees to have their

transactions mined more quickly. Since the blockchain is public, everyone can see the contents of others' pending transactions.

This means if a given user is revealing the solution to a puzzle or other valuable secret, a malicious user can steal the solution and copy their transaction with higher fees to preempt the original solution.

Test results: Having related vulnerabilities in smart contract code.

contracts/governance/CompLogic.sol

```
169     function approve(address spender, uint256 rawAmount)
170         external
171         returns (bool)
172     {
173         uint96 amount;
174         if (rawAmount == uint256(-1)) {
175             amount = uint96(-1);
176         } else {
177             amount = safe96(rawAmount, "Comp::approve: amount exceeds 96 bits");
178         }
179
180         allowances[msg.sender][spender] = amount;
181
182         emit Approval(msg.sender, spender, amount);
183         return true;
184     }
```

Safety advice:

1. User A allows the number of user B transfers to be N (N > 0) by calling the approve function;

2. After a while, user A decided to change N to M (M > 0), so he called the approve function again;

3. User B quickly calls the transfer from function to transfer the number of N before the second call is processed by the miner. After user A's second call to approve is successful, user B can get the transfer amount of M again. That is, user B obtains the transfer amount of N+M by trading sequence attack.

3.7. Denial of service attack detection【Pass】

In the blockchain world, denial of service is deadly, and smart contracts under attack of this type may never be able to return to normal. There may be a number of

reasons for a denial of service in smart contracts, including malicious behavior as a recipient of transactions, gas depletion caused by artificially increased computing gas, and abuse of access control to access the private components of the intelligent contract. Take advantage of confusion and neglect, etc.

Test results: No related vulnerabilities in smart contract code.

Safety advice：None.

3.8. Logical design Flaw【Pass】

Detect the security problems related to business design in the contract code.

Test results: No related vulnerabilities in smart contract code.

Safety advice: None.

3.9. USDT Fake Deposit Issue【Pass】

In the transfer function of the token contract, the balance check of the transfer initiator (msg.sender) is judged by if. When balances[msg.sender] < value, it enters the else logic part and returns false, and finally no exception is thrown. We believe that only the modest judgment of if/else is an imprecise coding method in the sensitive function scene such as transfer.

Detection results: No related vulnerabilities in smart contract code.

Safety advice: None.

3.10. Adding tokens【Pass】

It is detected whether there is a function in the token contract that may increase the total amount of tokens after the total amount of tokens is initialized.

Test results: No related vulnerabilities in smart contract code.

Safety advice：None.

3.11. Freezing accounts bypassed【Pass】

In the token contract, when transferring the token, it is detected whether there is an unverified token source account, an originating account, and whether the target account is frozen.

Detection results: No related vulnerabilities in smart contract code.

Safety advice: None.

4. Appendix A：Contract code

```
pragma solidity >=0.4.21 <0.7.0;
pragma experimental ABIEncoderV2;

import "@openzeppelin/upgrades/contracts/Initializable.sol";
import "../Token/TokenLogic.sol";
import "../governance/CompLogic.sol";


contract BridgeLogic is Initializable {
    using SafeMath for uint256;

    CompLogic public mpond;
    TokenLogic public pond;
    address owner;
    address governanceProxy;

    uint256 pondPerMpond;
    uint256 epochLength;
    uint256 startEpoch;
    uint256 startTime;

    uint256 lockTime;

    uint256 liquidityBp;
    uint256 liquidityDecimals;
    uint256 liquidityEpochLength;
    uint256 liquidityStartTime;
    uint256 liquidityStartEpoch;

    struct Requests {
        uint256 amount;
        uint256 releaseEpoch;
    }
    mapping(address => mapping(uint256 => Requests)) requests;
//address->epoch->Request(amount, lockTime)
    mapping(address => mapping(uint256 => uint256)) claimedAmounts;
//address->epoch->amount

    function initialize(
        address _mpond,
        address _pond,
        address _owner,
        address _governanceProxy
    ) public initializer {
        createConstants();
        mpond = CompLogic(_mpond);
        pond = TokenLogic(_pond);
        owner = _owner;
        governanceProxy = _governanceProxy;
        startTime = block.timestamp;
        liquidityStartTime = block.timestamp;
    }

    function createConstants() internal {
        pondPerMpond = 1000000;
        epochLength = 1 days;
        startEpoch = 0;
        lockTime = 180 days;
        liquidityBp = 20;
        liquidityDecimals = 4; // 0.0002 (i.e) 0.2%
        liquidityEpochLength = 180 days; // liquidty will increase arithmatically after
locktime
        liquidityStartEpoch = 1; //after locktime liquidity counter start from 1
    }

    function getConversionRate() public view returns (uint256) {
        return pondPerMpond;
    }

    function changeLiquidityBp(uint256 _newLbp) external returns (bool) {
        require(
            msg.sender == owner || msg.sender == governanceProxy,
```

```
        "Liquidity can be only changed by governance or owner"
    );
    liquidityBp = _newLbp;
    return true;
}

function getCurrentEpoch() internal view returns (uint256) {
    return (block.timestamp - startTime).div(epochLength) + startEpoch;
}

function lockTimeEpoch(uint256 _time) internal view returns (uint256) {
    return _time.div(epochLength);
}

function getLiquidityEpoch() public view returns (uint256) {
    if (block.timestamp < liquidityStartTime + 180 days) {
        return 0;
    }
    return
        (block.timestamp - liquidityStartTime - 180 days).div(
            liquidityEpochLength
        ) + liquidityStartEpoch;
}

function effectiveLiquidity() public view returns (uint256) {
    uint256 effective = getLiquidityEpoch().mul(liquidityBp);
    if (effective > 10000) {
        return 10000;
    } else {
        return effective;
    }
}

function getConvertableAmount(address _address, uint256 _epoch)
    public
    view
    returns (uint256)
{
    Requests memory _req = requests[_address][_epoch];
    uint256 _claimedAmount = claimedAmounts[_address][_epoch];
    if (
        _claimedAmount >= _req.amount.mul(effectiveLiquidity()).div(10000)
    ) {
        return 0;
    } else {
        return
            (_req.amount.mul(effectiveLiquidity()).div(10000)).sub(
                _claimedAmount
            );
    }
}

function getClaimedAmount(address _address, uint256 _epoch)
    public
    view
    returns (uint256)
{
    return claimedAmounts[_address][_epoch];
}

function convert(uint256 _epoch, uint256 _amount) public returns (uint256) {
    require(_amount > 0, "Should be non zero amount");
    uint256 _claimedAmount = claimedAmounts[msg.sender][_epoch];
    uint256 totalUnlockableAmount = _claimedAmount + _amount;
    Requests memory _req = requests[msg.sender][_epoch];
    require(
        totalUnlockableAmount <=
            _req.amount.mul(effectiveLiquidity()).div(10000),
        "total unlock amount should be less than requests_amount*effective_liquidity"
    );
    require(
        getCurrentEpoch() >= _req.releaseEpoch,
        "Funds can only be released after requests exceed locktime"
    );
    claimedAmounts[msg.sender][_epoch] = totalUnlockableAmount;
    mpond.transferFrom(msg.sender, address(this), _amount);
    pond.transfer(msg.sender, _amount.mul(pondPerMpond));
```

```
        return _amount.mul(pondPerMpond);
    }

    function placeRequest(uint256 amount) external returns (uint256, uint256) {
        uint256 epoch = getCurrentEpoch();
        require(amount != 0, "Request should be placed with non zero amount");
        require(
            requests[msg.sender][epoch].amount == 0,
            "Only one request per epoch is acceptable"
        );
        require(
            mpond.balanceOf(msg.sender) > 0,
            "mPond balance should be greated than 0 for placing requests"
        );
        require(
            amount <= mpond.balanceOf(msg.sender),
            "request should be placed with amount less than or equal to balance"
        );
        Requests memory _req = Requests(
            amount,
            epoch + lockTimeEpoch(lockTime)
        );
        requests[msg.sender][epoch] = _req;
        return (epoch, epoch + lockTimeEpoch(lockTime));
    }

    function viewRequest(address _address, uint256 _epoch)
        public
        view
        returns (Requests memory)
    {
        return requests[_address][_epoch];
    }

    function addLiquidity(uint256 _mpond, uint256 _pond)
        external
        returns (bool)
    {
        require(
            msg.sender == owner,
            "addLiquidity: only owner can call this function"
        );
        mpond.transferFrom(msg.sender, address(this), _mpond);
        pond.transferFrom(msg.sender, address(this), _pond);
        return true;
    }

    function removeLiquidity(
        uint256 _mpond,
        uint256 _pond,
        address _withdrawAddress
    ) external returns (bool) {
        require(
            msg.sender == owner,
            "removeLiquidity: only owner can call this function"
        );
        mpond.transfer(_withdrawAddress, _mpond);
        pond.transfer(_withdrawAddress, _pond);
        return true;
    }

    function getLiquidity() public view returns (uint256, uint256) {
        return (pond.balanceOf(address(this)), mpond.balanceOf(address(this)));
    }

    function getMpond(uint256 _mpond) public returns (uint256) {
        uint256 pondToDeduct = mpond.balanceOf(msg.sender).mul(pondPerMpond);
        pond.transferFrom(msg.sender, address(this), pondToDeduct);
        mpond.transfer(msg.sender, _mpond);
        return pondToDeduct;
    }
}


pragma solidity >=0.4.21 <0.7.0;
pragma experimental ABIEncoderV2;
import "@openzeppelin/upgrades/contracts/Initializable.sol";
```

```
contract CompLogic is Initializable {
    /// @notice EIP-20 token name for this token
    string public name;

    /// @notice EIP-20 token symbol for this token
    string public symbol;

    /// @notice EIP-20 token decimals for this token
    uint8 public decimals;

    /// @notice Total number of tokens in circulation
    uint256 public totalSupply; // 10k mPond
    uint256 public bridgeSupply; // 3k mPond
    /// @notice Allowance amounts on behalf of others
    mapping(address => mapping(address => uint96)) internal allowances;

    /// @notice Official record of token balances for each account
    mapping(address => uint96) internal balances;

    /// @notice A record of each accounts delegate
    mapping(address => mapping(address => uint96)) public delegates;

    /// @notice A checkpoint for marking number of votes from a given block
    struct Checkpoint {
        uint32 fromBlock;
        uint96 votes;
    }

    /// @notice A record of votes checkpoints for each account, by index
    mapping(address => mapping(uint32 => Checkpoint)) public checkpoints;

    /// @notice The number of checkpoints for each account
    mapping(address => uint32) public numCheckpoints;

    /// @notice The EIP-712 typehash for the contract's domain
    bytes32 public DOMAIN_TYPEHASH;

    /// @notice The EIP-712 typehash for the delegation struct used by the contract
    bytes32 public DELEGATION_TYPEHASH;

    /// @notice The EIP-712 typehash for the delegation struct used by the contract
    bytes32 public UNDELEGATION_TYPEHASH;

    /// @notice A record of states for signing / validating signatures
    mapping(address => uint256) public nonces;

    /// customized params
    address public admin;
    mapping(address => bool) public isWhiteListed;
    bool public enableAllTranfers;

    /// @notice An event thats emitted when an account changes its delegate
    event DelegateChanged(
        address indexed delegator,
        address indexed fromDelegate,
        address indexed toDelegate
    );

    /// @notice An event thats emitted when a delegate account's vote balance changes
    event DelegateVotesChanged(
        address indexed delegate,
        uint256 previousBalance,
        uint256 newBalance
    );

    /// @notice The standard EIP-20 transfer event
    event Transfer(address indexed from, address indexed to, uint256 amount);

    /// @notice The standard EIP-20 approval event
    event Approval(
        address indexed owner,
        address indexed spender,
        uint256 amount
    );
```

```
    /**
     * @notice Initializer a new Comp token
     * @param account The initial account to grant all the tokens
     */
    function initialize(address account, address bridge) public initializer {
        createConstants();
        require(
            account != bridge,
            "Bridge and accoutn should not be the same address"
        );
        balances[bridge] = uint96(bridgeSupply);
        delegates[bridge][address(0)] = uint96(bridgeSupply);
        isWhiteListed[bridge] = true;
        emit Transfer(address(0), bridge, bridgeSupply);

        uint96 remainingSupply = sub96(
            uint96(totalSupply),
            uint96(bridgeSupply),
            "Comp: Subtraction overflow in the constructor"
        );
        balances[account] = remainingSupply;
        delegates[account][address(0)] = remainingSupply;
        isWhiteListed[account] = true;
        emit Transfer(address(0), account, uint256(remainingSupply));
    }

    function createConstants() internal {
        name = "Marlin Governance Token";
        symbol = "mPOND";
        decimals = 18;
        totalSupply = 10000e18;
        bridgeSupply = 7000e18;
        DOMAIN_TYPEHASH = keccak256(
            "EIP712Domain(string name,uint256 chainId,address verifyingContract)"
        );
        DELEGATION_TYPEHASH = keccak256(
            "Delegation(address delegatee,uint256 nonce,uint256 expiry,uint96 amount)"
        );
        UNDELEGATION_TYPEHASH = keccak256(
            "Unelegation(address delegatee,uint256 nonce,uint256 expiry,uint96 amount)"
        );
        enableAllTranfers = true;
    }

    function addWhiteListAddress(address _address) external returns (bool) {
        require(msg.sender == admin, "Only admin can whitelist");
        isWhiteListed[_address] = true;
        return true;
    }

    function enableAllTransfers() external returns (bool) {
        require(msg.sender == admin, "Only enable can enable all transfers");
        enableAllTranfers = true;
        return true;
    }

    function isWhiteListedTransfer(address _address1, address _address2)
        public
        view
        returns (bool)
    {
        return
            (isWhiteListed[_address1] || isWhiteListed[_address2]) ||
            enableAllTranfers;
    }

    /**
     * @notice Get the number of tokens `spender` is approved to spend on behalf of
`account`
     * @param account The address of the account holding the funds
     * @param spender The address of the account spending the funds
     * @return The number of tokens approved
     */
    function allowance(address account, address spender)
        external
        view
        returns (uint256)
```

```
    {
        return allowances[account][spender];
    }

    /**
     * @notice Approve `spender` to transfer up to `amount` from `src`
     * @dev This will overwrite the approval amount for `spender`
     *  and is subject to issues noted
[here](https://eips.ethereum.org/EIPS/eip-20#approve)
     * @param spender The address of the account which may transfer tokens
     * @param rawAmount The number of tokens that are approved (2^256-1 means infinite)
     * @return Whether or not the approval succeeded
     */
    function approve(address spender, uint256 rawAmount)
        external
        returns (bool)
    {
        uint96 amount;
        if (rawAmount == uint256(-1)) {
            amount = uint96(-1);
        } else {
            amount = safe96(rawAmount, "Comp::approve: amount exceeds 96 bits");
        }

        allowances[msg.sender][spender] = amount;

        emit Approval(msg.sender, spender, amount);
        return true;
    }

    /**
     * @notice Get the number of tokens held by the `account`
     * @param account The address of the account to get the balance of
     * @return The number of tokens held
     */
    function balanceOf(address account) external view returns (uint256) {
        return balances[account];
    }

    /**
     * @notice Transfer `amount` tokens from `msg.sender` to `dst`
     * @param dst The address of the destination account
     * @param rawAmount The number of tokens to transfer
     * @return Whether or not the transfer succeeded
     */
    function transfer(address dst, uint256 rawAmount) external returns (bool) {
        require(
            isWhiteListedTransfer(msg.sender, dst),
            "Atleast of the address (msg.sender or dst) should be whitelisted"
        );
        uint96 amount = safe96(
            rawAmount,
            "Comp::transfer: amount exceeds 96 bits"
        );
        _transferTokens(msg.sender, dst, amount);
        return true;
    }

    /**
     * @notice Transfer `amount` tokens from `src` to `dst`
     * @param src The address of the source account
     * @param dst The address of the destination account
     * @param rawAmount The number of tokens to transfer
     * @return Whether or not the transfer succeeded
     */
    function transferFrom(
        address src,
        address dst,
        uint256 rawAmount
    ) external returns (bool) {
        require(
            isWhiteListedTransfer(msg.sender, dst),
            "Atleast of the address (src or dst) should be whitelisted"
        );
        address spender = msg.sender;
        uint96 spenderAllowance = allowances[src][spender];
        uint96 amount = safe96(
```

```
            rawAmount,
            "Comp::approve: amount exceeds 96 bits"
        );

        if (spender != src && spenderAllowance != uint96(-1)) {
            uint96 newAllowance = sub96(
                spenderAllowance,
                amount,
                "Comp::transferFrom: transfer amount exceeds spender allowance"
            );
            allowances[src][spender] = newAllowance;

            emit Approval(src, spender, newAllowance);
        }

        _transferTokens(src, dst, amount);
        return true;
    }

    /**
     * @notice Delegate votes from `msg.sender` to `delegatee`
     * @param delegatee The address to delegate votes to
     */
    function delegate(address delegatee, uint96 amount) public {
        return _delegate(msg.sender, delegatee, amount);
    }

    function undelegate(address delegatee, uint96 amount) public {
        return _undelegate(msg.sender, delegatee, amount);
    }

    /**
     * @notice Delegates votes from signatory to `delegatee`
     * @param delegatee The address to delegate votes to
     * @param nonce The contract state required to match the signature
     * @param expiry The time at which to expire the signature
     * @param v The recovery byte of the signature
     * @param r Half of the ECDSA signature pair
     * @param s Half of the ECDSA signature pair
     */
    function delegateBySig(
        address delegatee,
        uint256 nonce,
        uint256 expiry,
        uint8 v,
        bytes32 r,
        bytes32 s,
        uint96 amount
    ) public {
        bytes32 domainSeparator = keccak256(
            abi.encode(
                DOMAIN_TYPEHASH,
                keccak256(bytes(name)),
                getChainId(),
                address(this)
            )
        );
        bytes32 structHash = keccak256(
            abi.encode(DELEGATION_TYPEHASH, delegatee, nonce, expiry, amount)
        );
        bytes32 digest = keccak256(
            abi.encodePacked("\x19\x01", domainSeparator, structHash)
        );
        address signatory = ecrecover(digest, v, r, s);
        require(
            signatory != address(0),
            "Comp::delegateBySig: invalid signature"
        );
        require(
            nonce == nonces[signatory]++,
            "Comp::delegateBySig: invalid nonce"
        );
        require(now <= expiry, "Comp::delegateBySig: signature expired");
        return _delegate(signatory, delegatee, amount);
    }

    function undelegateBySig(
```

```
        address delegatee,
        uint256 nonce,
        uint256 expiry,
        uint8 v,
        bytes32 r,
        bytes32 s,
        uint96 amount
    ) public {
        bytes32 domainSeparator = keccak256(
            abi.encode(
                DOMAIN_TYPEHASH,
                keccak256(bytes(name)),
                getChainId(),
                address(this)
            )
        );
        bytes32 structHash = keccak256(
            abi.encode(UNDELEGATION_TYPEHASH, delegatee, nonce, expiry, amount)
        );
        bytes32 digest = keccak256(
            abi.encodePacked("\x19\x01", domainSeparator, structHash)
        );
        address signatory = ecrecover(digest, v, r, s);
        require(
            signatory != address(0),
            "Comp::undelegateBySig: invalid signature"
        );
        require(
            nonce == nonces[signatory]++,
            "Comp::undelegateBySig: invalid nonce"
        );
        require(now <= expiry, "Comp::undelegateBySig: signature expired");
        return _undelegate(signatory, delegatee, amount);
    }

    /**
     * @notice Gets the current votes balance for `account`
     * @param account The address to get votes balance
     * @return The number of current votes for `account`
     */
    function getCurrentVotes(address account) external view returns (uint96) {
        uint32 nCheckpoints = numCheckpoints[account];
        return
            nCheckpoints > 0 ? checkpoints[account][nCheckpoints - 1].votes : 0;
    }

    /**
     * @notice Determine the prior number of votes for an account as of a block number
     * @dev Block number must be a finalized block or else this function will revert to
prevent misinformation.
     * @param account The address of the account to check
     * @param blockNumber The block number to get the vote balance at
     * @return The number of votes the account had as of the given block
     */
    function getPriorVotes(address account, uint256 blockNumber)
        public
        view
        returns (uint96)
    {
        require(
            blockNumber < block.number,
            "Comp::getPriorVotes: not yet determined"
        );

        uint32 nCheckpoints = numCheckpoints[account];
        if (nCheckpoints == 0) {
            return 0;
        }

        // First check most recent balance
        if (checkpoints[account][nCheckpoints - 1].fromBlock <= blockNumber) {
            return checkpoints[account][nCheckpoints - 1].votes;
        }

        // Next check implicit zero balance
        if (checkpoints[account][0].fromBlock > blockNumber) {
            return 0;
```

```
        }

        uint32 lower = 0;
        uint32 upper = nCheckpoints - 1;
        while (upper > lower) {
            uint32 center = upper - (upper - lower) / 2; // ceil, avoiding overflow
            Checkpoint memory cp = checkpoints[account][center];
            if (cp.fromBlock == blockNumber) {
                return cp.votes;
            } else if (cp.fromBlock < blockNumber) {
                lower = center;
            } else {
                upper = center - 1;
            }
        }
        return checkpoints[account][lower].votes;
    }

    function _delegate(
        address delegator,
        address delegatee,
        uint96 amount
    ) internal {
        delegates[delegator][address(0)] = sub96(
            delegates[delegator][address(0)],
            amount,
            "Comp: delegates underflow"
        );
        delegates[delegator][delegatee] = add96(
            delegates[delegator][delegatee],
            amount,
            "Comp: delegates overflow"
        );

        emit DelegateChanged(delegator, address(0), delegatee);

        _moveDelegates(address(0), delegatee, amount);
    }

    function _undelegate(
        address delegator,
        address delegatee,
        uint96 amount
    ) internal {
        delegates[delegator][delegatee] = sub96(
            delegates[delegator][delegatee],
            amount,
            "Comp: undelegates underflow"
        );
        delegates[delegator][address(0)] = add96(
            delegates[delegator][address(0)],
            amount,
            "Comp: delegates underflow"
        );
        emit DelegateChanged(delegator, delegatee, address(0));
        _moveDelegates(delegatee, address(0), amount);
    }

    function _transferTokens(
        address src,
        address dst,
        uint96 amount
    ) internal {
        require(
            src != address(0),
            "Comp::_transferTokens: cannot transfer from the zero address"
        );
        require(
            delegates[src][address(0)] >= amount,
            "Comp: _transferTokens: undelegated amount should be greater than transfer
amount"
        );
        require(
            dst != address(0),
            "Comp::_transferTokens: cannot transfer to the zero address"
        );
```

```
        balances[src] = sub96(
            balances[src],
            amount,
            "Comp::_transferTokens: transfer amount exceeds balance"
        );
        delegates[src][address(0)] = sub96(
            delegates[src][address(0)],
            amount,
            "Comp: _tranferTokens: undelegate subtraction error"
        );

        balances[dst] = add96(
            balances[dst],
            amount,
            "Comp::_transferTokens: transfer amount overflows"
        );
        delegates[dst][address(0)] = add96(
            delegates[dst][address(0)],
            amount,
            "Comp: _transferTokens: undelegate addition error"
        );
        emit Transfer(src, dst, amount);

        // _moveDelegates(delegates[src], delegates[dst], amount);
    }

    function _moveDelegates(
        address srcRep,
        address dstRep,
        uint96 amount
    ) internal {
        if (srcRep != dstRep && amount > 0) {
            if (srcRep != address(0)) {
                uint32 srcRepNum = numCheckpoints[srcRep];
                uint96 srcRepOld = srcRepNum > 0
                    ? checkpoints[srcRep][srcRepNum - 1].votes
                    : 0;
                uint96 srcRepNew = sub96(
                    srcRepOld,
                    amount,
                    "Comp::_moveVotes: vote amount underflows"
                );
                _writeCheckpoint(srcRep, srcRepNum, srcRepOld, srcRepNew);
            }

            if (dstRep != address(0)) {
                uint32 dstRepNum = numCheckpoints[dstRep];
                uint96 dstRepOld = dstRepNum > 0
                    ? checkpoints[dstRep][dstRepNum - 1].votes
                    : 0;
                uint96 dstRepNew = add96(
                    dstRepOld,
                    amount,
                    "Comp::_moveVotes: vote amount overflows"
                );
                _writeCheckpoint(dstRep, dstRepNum, dstRepOld, dstRepNew);
            }
        }
    }

    function _writeCheckpoint(
        address delegatee,
        uint32 nCheckpoints,
        uint96 oldVotes,
        uint96 newVotes
    ) internal {
        uint32 blockNumber = safe32(
            block.number,
            "Comp::_writeCheckpoint: block number exceeds 32 bits"
        );

        if (
            nCheckpoints > 0 &&
            checkpoints[delegatee][nCheckpoints - 1].fromBlock == blockNumber
        ) {
            checkpoints[delegatee][nCheckpoints - 1].votes = newVotes;
        } else {
```

```
            checkpoints[delegatee][nCheckpoints] = Checkpoint(
                blockNumber,
                newVotes
            );
            numCheckpoints[delegatee] = nCheckpoints + 1;
        }

        emit DelegateVotesChanged(delegatee, oldVotes, newVotes);
    }

    function safe32(uint256 n, string memory errorMessage)
        internal
        pure
        returns (uint32)
    {
        require(n < 2**32, errorMessage);
        return uint32(n);
    }

    function safe96(uint256 n, string memory errorMessage)
        internal
        pure
        returns (uint96)
    {
        require(n < 2**96, errorMessage);
        return uint96(n);
    }

    function add96(
        uint96 a,
        uint96 b,
        string memory errorMessage
    ) internal pure returns (uint96) {
        uint96 c = a + b;
        require(c >= a, errorMessage);
        return c;
    }

    function sub96(
        uint96 a,
        uint96 b,
        string memory errorMessage
    ) internal pure returns (uint96) {
        require(b <= a, errorMessage);
        return a - b;
    }

    function getChainId() internal pure returns (uint256) {
        uint256 chainId;
        assembly {
            chainId := chainid()
        }
        return chainId;
    }
}


pragma solidity >=0.4.21 <0.7.0;

import "./SafeMath.sol";
import "../governance/Comp.sol";
import "solidity-bytes-utils/contracts/BytesLib.sol";


contract StakeDrop {
    using SafeMath for uint256;
    using BytesLib for bytes;

    struct Signature {
        bytes32 hash;
        uint8 v;
        bytes32 r;
        bytes32 s;
    }

    struct WithdrawlParams {
        uint32 chainType;
```

```
    bytes32 stakingAddressHash;
    uint32 currentWithdrawlId;
    address mappedAddress;
    uint256 withdrawlAmount;
}

address admin;
address signer;
Comp comp;
mapping(address => uint256) etherDeposits;
mapping(address => uint256) withdrawnBalances;
//chainNumber -> address -> externalNonce
mapping(uint32 => mapping(bytes32 => uint32)) withdrawlCounts;

event Withdraw(
    uint32 indexed chainType,
    bytes32 indexed stakingAddressHash,
    uint32 indexed withdrawlId,
    uint256 value
);

constructor(address _signer, address _tokenAddress) public {
    admin = msg.sender;
    signer = _signer;
    comp = Comp(_tokenAddress);
}

function deposit() public payable {
    etherDeposits[msg.sender].add(msg.value);
}

function depositTokens(uint256 amount) public {
    comp.transferFrom(msg.sender, address(this), amount);
}

function withdraw(address _address, uint256 _amount) public {
    require(msg.sender == signer, "Should be only called by signer");
    comp.transfer(_address, _amount);
}

function withdrawUsingSig(bytes calldata _bytes) external returns (bool) {
    WithdrawlParams memory w = extractBytes(_bytes);
    Signature memory sig = getSignature(_bytes);
    address _recovered = ecrecover(sig.hash, sig.v, sig.r, sig.s);

    uint32 lastWithdrawlId = withdrawlCounts[w.chainType][w
        .stakingAddressHash];

    require(_recovered == signer, "msg should be generated from signer");
    require(
        w.currentWithdrawlId == lastWithdrawlId + 1,
        "withdrawls must be sequential"
    );
    require(
        w.mappedAddress == msg.sender,
        "Only sender can claim his rewards"
    );
    withdrawlCounts[w.chainType][w.stakingAddressHash] = w
        .currentWithdrawlId;
    comp.transfer(msg.sender, w.withdrawlAmount);
    withdrawnBalances[msg.sender] = withdrawnBalances[msg.sender].add(
        w.withdrawlAmount
    );
    emit Withdraw(
        w.chainType,
        w.stakingAddressHash,
        w.currentWithdrawlId,
        withdrawnBalances[msg.sender]
    );
    return true;
}

function getWithdrawnBalance(address _address)
    public
    view
    returns (uint256)
{
```

```
        return withdrawnBalances[_address];
    }

    function extractBytes(bytes memory _bytes)
        internal
        pure
        returns (WithdrawlParams memory)
    {
        uint32 chainType = _bytes.slice(0, 4).toUint32(0);
        bytes32 stakingAddressHash = _bytes.toBytes32(4);
        uint32 currentWithdrawlId = _bytes.toUint32(36);
        address mappedAddress = _bytes.toAddress(40);
        uint256 withdrawlAmount = _bytes.toUint(60);
        WithdrawlParams memory w = WithdrawlParams(
            chainType,
            stakingAddressHash,
            currentWithdrawlId,
            mappedAddress,
            withdrawlAmount
        );
        return w;
    }

    // function withdrawUsingSig(bytes calldata _bytes)
    //     external
    //     returns (
    //         uint32,
    //         bytes32,
    //         uint32,
    //         uint32,
    //         address,
    //         uint256
    //     )
    // {
    //     uint32 chainType = _bytes.slice(0, 4).toUint32(0);
    //     bytes32 stakingAddressHash = _bytes.toBytes32(4);
    //     uint32 currentWithdrawlId = _bytes.toUint32(36);
    //     uint32 lastWithdrawlId = withdrawlCounts[chainType][stakingAddressHash];
    //     address mappedAddress = _bytes.toAddress(40);
    //     uint256 withdrawlAmount = _bytes.toUint(60);
    //     Signature memory sig = getSignature(_bytes);
    //     require(
    //         currentWithdrawlId == lastWithdrawlId + 1,
    //         "withdrawls must be sequential"
    //     );
    //     require(
    //         mappedAddress == msg.sender,
    //         "Only sender can claim his rewards"
    //     );
    //     emit Withdraw(chainType, stakingAddressHash, currentWithdrawlId);
    //     return (
    //         chainType,
    //         stakingAddressHash,
    //         currentWithdrawlId,
    //         lastWithdrawlId,
    //         mappedAddress,
    //         withdrawlAmount
    //     );
    // }
    // function withdrawUsingSig(bytes calldata _bytes)
    //     external
    //     returns (
    //         bytes32,
    //         uint8,
    //         bytes32,
    //         bytes32,
    //         address
    //     )
    // {
    //     uint32 chainType = _bytes.slice(0, 4).toUint32(0);
    //     bytes32 stakingAddressHash = _bytes.toBytes32(4);
    //     uint32 currentWithdrawlId = _bytes.toUint32(36);
    //     uint32 lastWithdrawlId = withdrawlCounts[chainType][stakingAddressHash];
    //     address mappedAddress = _bytes.toAddress(40);
    //     uint256 withdrawlAmount = _bytes.toUint(60);

    //     Signature memory sig = getSignature(_bytes);
```

```
        //    address _recoverd = ecrecover(sig.hash, sig.v, sig.r, sig.s);

        //    // require(
        //    //    signer == _recoverd,
        //    //    "Message should be signed only by authorized signer outside the chain"
        //    // );

        //    require(
        //        currentWithdrawlId == lastWithdrawlId + 1,
        //        "withdrawls must be sequential"
        //    );
        //    require(
        //        mappedAddress == msg.sender,
        //        "Only sender can claim his rewards"
        //    );
        //    emit Withdraw(chainType, stakingAddressHash, currentWithdrawlId);
        //    return (sig.hash, sig.v, sig.r, sig.s, _recoverd);
        // }

        function getSignature(bytes memory _data)
            internal
            pure
            returns (Signature memory)
        {
            //Image prefix to be decided by the team
            bytes memory prefix = "\x19Ethereum Signed Message:\n32";
            bytes32 messageHash = keccak256(abi.encodePacked(_data.slice(0, 92)));
            bytes32 _hash1 = keccak256(abi.encodePacked(prefix, messageHash));
            uint8 _v1 = _data.slice(92, 1).toUint8(0);
            bytes32 _r1 = _data.slice(93, 32).toBytes32(0);
            bytes32 _s1 = _data.slice(125, 32).toBytes32(0);
            Signature memory sig = Signature(_hash1, _v1, _r1, _s1);
            return sig;
        }
    }


    pragma solidity >=0.4.21 <0.7.0;

    import "@openzeppelin/upgrades/contracts/Initializable.sol";
    import "@openzeppelin/contracts-ethereum-package/contracts/token/ERC20/ERC20.sol";
    import
"@openzeppelin/contracts-ethereum-package/contracts/token/ERC20/ERC20Detailed.sol";
    // import
"@openzeppelin/contracts-ethereum-package/contracts/token/ERC20/ERC20Mintable.sol";
    import
"@openzeppelin/contracts-ethereum-package/contracts/token/ERC20/ERC20Capped.sol";
    import
"@openzeppelin/contracts-ethereum-package/contracts/token/ERC20/ERC20Burnable.sol";


    contract TokenLogic is
        Initializable,
        ERC20,
        ERC20Detailed,
        // ERC20Mintable,
        ERC20Capped,
        ERC20Burnable
    {
        function initialize(
            string memory _name,
            string memory _symbol,
            uint8 _decimal,
            address _bridge
        ) public initializer {
            ERC20Detailed.initialize(_name, _symbol, _decimal);
            // ERC20Mintable.initialize(msg.sender);
            ERC20Capped.initialize(10000000000e18, msg.sender);
            mint(_bridge, 3000000000e18);
        }
    }


    pragma solidity >=0.4.21 <0.7.0;


    /// @title Contract to reward overlapping stakes
```

```
    /// @author Marlin
    /// @notice Use this contract only for testing
    /// @dev Contract may or may not change in future (depending upon the new slots in
proxy-store)
    contract TokenProxy {
        bytes32 internal constant IMPLEMENTATION_SLOT = bytes32(
            uint256(keccak256("eip1967.proxy.implementation")) - 1
        );
        bytes32 internal constant PROXY_ADMIN_SLOT = bytes32(
            uint256(keccak256("eip1967.proxy.admin")) - 1
        );

        constructor(address contractLogic) public {
            // save the code address
            bytes32 slot = IMPLEMENTATION_SLOT;
            assembly {
                sstore(slot, contractLogic)
            }
            // save the proxy admin
            slot = PROXY_ADMIN_SLOT;
            address sender = msg.sender;
            assembly {
                sstore(slot, sender)
            }
        }

        /// @author Marlin
        /// @dev Only admin can update the contract
        /// @param _newLogic address is the address of the contract that has to updated to
        function updateLogic(address _newLogic) public {
            require(
                msg.sender == getAdmin(),
                "Only Admin should be able to update the contracts"
            );
            bytes32 slot = IMPLEMENTATION_SLOT;
            assembly {
                sstore(slot, _newLogic)
            }
        }

        /// @author Marlin
        /// @dev use assembly as contract store slot is manually decided
        function getAdmin() internal view returns (address result) {
            bytes32 slot = PROXY_ADMIN_SLOT;
            assembly {
                result := sload(slot)
            }
        }

        /// @author Marlin
        /// @dev add functionality to forward the balance as well.
        function() external payable {
            bytes32 slot = IMPLEMENTATION_SLOT;
            assembly {
                let contractLogic := sload(slot)
                calldatacopy(0x0, 0x0, calldatasize())
                let success := delegatecall(
                    sub(gas(), 10000),
                    contractLogic,
                    0x0,
                    calldatasize(),
                    0,
                    0
                )
                let retSz := returndatasize()
                returndatacopy(0, 0, retSz)

                switch success
                    case 0 {
                        revert(0, retSz)
                    }
                    default {
                        return(0, retSz)
                    }
            }
        }
    }
```

5. Appendix B: vulnerability risk rating criteria

Smart contract vulnerability rating standard	
Vulnerability rating	Vulnerability rating description
High risk vulnerability	The loophole which can directly cause the contract or the user's fund loss, such as the value overflow loophole which can cause the value of the substitute currency to zero, the false recharge loophole that can cause the exchange to lose the substitute coin, can cause the contract account to lose the ETH or the reentry loophole of the substitute currency, and so on; It can cause the loss of ownership rights of token contract, such as: the key function access control defect or call injection leads to the key function access control bypassing, and the loophole that the token contract can not work properly. Such as: a denial-of-service vulnerability due to sending ETHs to a malicious address, and a denial-of-service vulnerability due to gas depletion.
Middle risk vulnerability	High risk vulnerabilities that need specific addresses to trigger, such as numerical overflow vulnerabilities that can be triggered by the owner of a token contract, access control defects of non-critical functions, and logical design defects that do not result in direct capital losses, etc.
Low risk vulnerability	A vulnerability that is difficult to trigger, or that will harm a limited number after triggering, such as a numerical overflow that requires a large number of ETH or tokens to trigger, and a vulnerability that the attacker cannot directly profit from after triggering a numerical overflow. Rely on risks by specifying the order of transactions triggered by a high gas.

6. Appendix C：Introduction of test tool

6.1. Manticore

Manticore is a symbolic execution tool for analysis of binaries and smart contracts. It discovers inputs that crash programs via memory safety violations. Manticore records an instruction-level trace of execution for each generated input and exposes programmatic access to its analysis engine via a Python API.

6.2. Oyente

Oyente is a smart contract analysis tool that Oyente can use to detect common bugs in smart contracts, such as reentrancy, transaction ordering dependencies, and more. More conveniently, Oyente's design is modular, so this allows advanced users to implement and insert their own detection logic to check for custom attributes in their contracts.

6.3. securify.sh

Securify can verify common security issues with smart contracts, such as transactional out-of-order and lack of input validation. It analyzes all possible execution paths of the program while fully automated. In addition, Securify has a specific language for specifying vulnerabilities. Securify can keep an eye on current security and other reliability issues.

6.4. Echidna

Echidna is a Haskell library designed for fuzzing EVM code.

6.5. MAIAN

MAIAN is an automated tool for finding smart contract vulnerabilities. Maian deals with the contract's bytecode and tries to establish a series of transactions to find and confirm errors.

6.6. ethersplay

Ethersplay is an EVM disassembler that contains related analysis tools.

6.7. ida-evm

Ida-evm is an IDA processor module for the Ethereum Virtual Machine (EVM).

6.8. Remix-ide

Remix is a browser-based compiler and IDE that allows users to build blockchain contracts and debug transactions using the Solidity language.

6.9. Knownsec Penetration Tester Special Toolkit

Knownsec penetration tester special tool kit, developed and collected by Knownsec penetration testing engineers, includes batch automatic testing tools dedicated to testers, self-developed tools, scripts, or utility tools.